Services

Overview

Services in the PlayPlus architecture handle data operations, business logic, and external API communication. They provide a clean abstraction layer between your application and external data sources, ensuring consistent data management across your application.

Key Characteristics

Data Layer Abstraction

- Handle all external API communication
- Provide consistent data interfaces
- Manage caching and state synchronization
- Handle error scenarios gracefully

State Management

- Use RxJS BehaviorSubject for reactive state
- Provide observable streams for data updates
- Implement caching strategies
- Support real-time data synchronization

Error Handling

- Comprehensive error handling
- Retry mechanisms
- User-friendly error messages
- Logging and monitoring support

Template Structure

TypeScript Service (service.ts.hbs)

Features

Built-in Features

- CRUD Operations Create, Read, Update, Delete operations Type-safe interfaces Consistent error handling
- Reactive State Management BehaviorSubject for current state Observable streams for updates Automatic cache updates
- Error Handling Comprehensive error catching Detailed error logging Graceful error propagation
- Caching In-memory data caching Cache invalidation strategies Cache clearing methods
- Type Safety Strongly typed interfaces Generic type support IntelliSense support

Best Practices

- Service Structure @ Injectable ({ providedIn : "root" , }) export class ExampleService { private apiUrl = "/api/example" ; private dataSubject = new BehaviorSubject < ExampleData [ ] > ([ ]) ; public data$ = this . dataSubject . asObservable ( ) ; }
- Error Handling getData ( ) : Observable < ExampleData [ ] > { return this . http . get < ExampleData [ ] > (this . apiUrl) . pipe (map (data => { this . dataSubject . next (data) ; return data ; }) , catchError (error => { console . error ('Error fetching data:' , error) ; throw error ; })) ; }
- Cache Management getCurrentData ( ) : ExampleData [ ] { return this . dataSubject . value ; } clearCache ( ) : void { this . dataSubject . next ([ ]) ; }

Usage Examples

Basic Service

This creates:

- UserService with CRUD operations
- UserData interface
- Comprehensive error handling
- Caching mechanisms

Advanced Service

This creates:

- ProductService with custom interface
- Advanced caching strategies

- Real-time data synchronization

Integration with Components

Container Components

Multiple Services

Data Interfaces

Basic Interface

Complex Interface

Advanced Patterns

Caching Strategies

Real-time Updates

Testing

Service Testing

Error Handling Strategies

Retry Logic

Error Transformation

Performance Optimization

Lazy Loading

Next Steps

After creating a service:

- Define the data interface for your specific domain
- Implement CRUD operations for your API endpoints
- Add error handling specific to your use case
- Implement caching strategies if needed
- Write comprehensive tests for all operations
- Integrate with container components for data flow

Developer Checklist

Before Creating Services:

- Are data interfaces defined for all entities?
- Is HTTP error handling implemented with meaningful messages?
- Is BehaviorSubject used for reactive state management?
- Are CRUD operations implemented for all entities?
- Is retry logic added for network failures?
- Are caching strategies implemented where needed?
- Are all API endpoints typed with TypeScript?
- Is data validation and sanitization implemented?
- Do I have unit tests with HTTP mocking?
- Is logging added for debugging and monitoring?
- Are RxJS operators used for async operations?
- Are cache invalidation strategies implemented?